
ERDC MSRC PET Technical Report No. 01-28

# STWAVE: A Case Study in Dual-Level Parallelism

by

Rebecca Fahey
Jane Smith

13 July 2001

**Work funded by the Department of Defense**
**High Performance Computing Modernization Program**
**U.S. Army Engineer Research and Development Center**
**Major Shared Resource Center through**

**Programming Environment and Training**

Supported by Contract Number: DAHC94-96-C0002
Computer Sciences Corporation

# STWAVE: A Case Study in Dual-Level Parallelism

Rebecca Fahey, Computer Sciences Corporation
U. S. Army Engineer Research and Development Center
Major Shared Resource Center(ERDC MSRC)
Jane Smith, Ph.D., Coastal and Hydraulics Laboratory
U. S. Army Engineer Research and Development Center

July 13, 2001

## 1 Introduction

This case study explores the dual-level parallel implementation of the Steady-State Spectral Wave Model (STWAVE) code[1]. STWAVE is a near-shore, wind-wave, growth and propagation simulation program. Two natural levels of parallelism exist in the model and are exploited in this work.

The calculations that process multiple wave runs are embarrassingly parallel and are distributed to the available MPI processes resulting in an almost linearly scalable MPI code. However, the number of MPI processes cannot be increased beyond the number of wave runs, limiting the achievable speedup with MPI. Moreover, within each MPI task, a number of lengthy loops are being repeatedly executed, and this natural second level of parallelism is exploited with OpenMP to obtain additional speedup.

Dual-level MPI/OpenMP parallel programming, has many advantages. For MPI or OpenMP programs with limited scalability and an existing second level of parallelism, the speedup attained surpasses the speedup possible with either method alone. The program is also more versatile because the hybrid code can be used to its full advantage on a shared memory architecture and still be used as an MPI code on a distributed architecture. These advantages are propelling dual-level parallel programming models to the forefront of high performance computing.

In this article, the development of a dual-level parallel program from the serial STWAVE code is explored. An overview of the STWAVE code along with an overview of the parallelization is given. Scalability issues are

[1]The serial version of STWAVE was developed as part of the Coastal Inlets Research Program by Donald T. Resio and Jane McKee Smith of the U.S. Army Engineer Research and Development Center (ERDC).

discussed and timings are presented from two high performance computers, an SGI Origin 2000 and an IBM Power3 SMP.

# 2 Overview of STWAVE [2]

STWAVE is an easy-to-apply, robust, steady-state model for nearshore, wind-wave growth and propagation. The model numerically solves the steady-state convervation of spectra action balance along backward-traced wave rays using finite difference methods. STWAVE is used routinely on coastal projects in the United States and abroad (over 40 recent or ongoing U.S. Corps of Engineers applications) to estimate wave heights, periods, and direction for projects related to sediment transport and navigation. The processes represented in STWAVE include refraction, shoaling, wave-current interaction, wave breaking, and wind-wave generation. Input into the model includes bathymetry, offshore spectra, water levels, and currents(optional). Assumptions made in STWAVE include mild bottom slope; spatially homogeneous offshore waves; steady waves, currents, and winds; linear refraction and shoaling; depth-uniform current; and negligible bottem friction. STWAVE has been incorporated into the Surface-Water Modeling System, which provides a user interface and supporting software for grid generation, interpolation of current fields, generation of input spectra, and visualization of model output.

Multiple wave model runs are executed by specifying multiple input wave spectra and current fields in the input files. During multiple runs, the model paramaters, and bathymetry, which are specified in other input files, remain constant. The calculations for each of these wave runs are independent, providing the basis for the MPI implementation of the code.

# 3 Overview of the Parallelization

MPI is used to distribute the calculations for each wave run to the available processes, while OpenMP is used to exploit the remaining loop level parallelism. The embarrassingly parallel calculations for each wave run are distributed to the available MPI processes resulting in a cyclic distribution of the spectra boundary and current field data. The bathymetry and model parameter data is replicated because it is used by each process. The calculations for each wave run are performed independently with no communication between MPI processes. Within each MPI task, OpenMP threads are used to parallelize several of the most time intensive loops.

## 3.1 MPI Implementation

The embarrassingly parallel calculations for each set of spectra boundary data are distributed to the available MPI processes. Each MPI processes reads its assigned data from an input file resulting in a cyclic distribution

of the spectra boundary data. Calculations are performed independently with no communication between MPI processes. The calculated results from each input set are stored in temporary files which must be assembled during post-processing.

For the MPI implementation, a method to distribute the spectra boundary and current data to the processors was required. Since processing each wave run required nearly the same amount of time, a master-slave approach was not deemed necessary. Instead, each processor reads through the data until it reaches the data needed for its assigned iteration. Data which is not needed for the assigned iteration is simply overwritten. To group the reads together, some minor code reorganization was necessary. Reading the input data could be done more efficiently if direct-access input files were used. However, such a change would require undesired changes to preprocessing programs.

To generate the same output files as the serial version, the MPI implementation utilizes temporary output files for each wave run. After STWAVE completes, a post-processing script is used to assemble the files. Scripts were constructed to automate this process. An execution script, runSTWAVE, is used to submit the process script to the batch system which runs STWAVE on the specified input data sets. A post-processing script is also queued to perform the post-processing after the initial job completes.

## 3.2 OpenMP Implementation

Within each MPI task, loop-level parallelism is also exploited. This parallelism could not be exploited well with MPI because the excessive communication necessitated by data dependencies would severely limit the gains in scalability. The use of OpenMP, which utilizes shared memory, allowed this additional level of parallelism without the communication overhead.

Since automatic, compiler generated threading produced little speedup, OpenMP directives were hand coded into the most time intensive subroutines. With only one exception, OpenMP was used exclusively to divide the iterations of large loops among threads. Before each of the nine most time intensive loops, a compiler directive was inserted to create/wake-up threads and specify the variable scoping. Also, in one subroutine, OpenMP directives were added to divide the subroutine into sections which could be executed in parallel.

The OpenMP directives comprise less than 50 lines of additional code, which can be activated at compile time if desired. No code modifications were necessary to implement the OpenMP. However, indices on some loops were switched and/or the loops slightly restructured for better cache utilization by avoiding invalidation of cache lines as the threads update shared arrays.

This type of programming has a tremendous amount of versitility. The single code can be compiled to produce: 1) a dual-level parallel program utilizing both MPI and OpenMP that can be used on shared memory archi-

tectures and hybrid architectures, 2) an MPI code for use on distributed architectures, 3) an OpenMP code that could be used when threading is desired but MPI is not, and 4) a serial version that could be used on workstations without parallel processing capabilities. Since this versitility complicates installation, a makefile system was designed to simplify the process. Platform specific makefiles are provided for several high performance computer platforms.

# 4 Scalability

The MPI portion scales to $n$, where $n$ is the number of wave runs to be completed. It will not scale beyond $n$ MPI processes since there are no additional wave runs to distribute to the extra processors. The efficiency[2] of the code remains good for small numbers of processors, but drops off as $n$ is approached due to the overhead associated with replicating data.

The speedup gained by threading was consistent on the two high performance computers tested. The speedup is dependent upon the grid size of the problem, with larger grid sizes exhibiting better thread scalability. Using a grid size of 301x511, a 40% decrease in execution time is observed on both machines utilizing two threads. Additional decreases are obtained for additional threads, with a 66% decrease obtained with 7 threads. Above seven threads, the amount of work assigned to each thread is insufficient to yield significant additional speedup in this test case.

## 4.1 MPI Scalability

When compiled with only MPI and no threads, the code scales almost linearly for appropriately chosen numbers of processors. Table 1 gives the execution times for 36-wave runs on an IBM Power3 SMP located at the Engineer Research and Development Center (ERDC) Major Shared Research Center (MSRC). The timings were conducted in a non-dedicated production environment. However, care was taken to insure sole access to all nodes. Also, since communication is not required by the program, jobs running on other nodes should have little effect on the execution time.

Table 1 shows nearly linear scalability when an appropriately chosen small number of processors are used. Given $n$-wave runs and $p$ processors, nearly linear scalability results for $\frac{n}{p}$ processors, as long as $p$ is less than half of $n$. Since the amount of replicated data is significant, the efficiency is better when the processors reuse the replicated data for several wave runs. When the processors are only assigned one or two wave runs, the overhead associated

[2]Efficiency is calculated using the formula $E = T_1/(P * T_P)$, where $T_1$ is the execution time for the serial program, $P$ is the number of processors, and $T_P$ is the execution time for the parallel version running on $P$ processors. Speedup is calculated using the formula $S = PE$. Both equation were taken from [1].

Execution Times for the MPI version
on an IBM Power3 SMP

| MPI Processes | Execution Time | Speedup | Efficiency |
|---|---|---|---|
| serial | 13hr15m | 1 | 100% |
| 2 | 6hr44m | 2.0 | 98.4% |
| 3 | 4hr32m | 2.9 | 97.4% |
| 6 | 2hr24m | 5.5 | 92.0% |
| 12 | 1hr16m | 10.5 | 87.2% |
| 18 | 55min | 14.5 | 80.3% |
| 36 | 33min | 24.1 | 66.9% |

Table 1: Execution times, relative speedup, and relative efficiency for various numbers of processors on the IBM Power3 SMP located at ERDC MSRC.

with the replicated data becomes a significant portion of the runtime and adversely effects the efficiency.

It is important for load balancing that $n$ be divisible by $p$. This insures that each processor is assigned the same number of wave runs. When it is not possible to choose $p$ such that $p$ divides $n$, the best load balancing will be achieved by choosing $p$ such that $p$ divides $n + e$ where $e$ is the smallest possible integer. This is important because $e$ processors will be idle while the other processors complete their last assigned wave run. If $e$ is small and each processor is assigned several wave runs, the idle time should not become a concern.

The efficiency can be maintained at an acceptable level by choosing an appropriate number of MPI processes. Also, when speedup is more important than the efficient use of computer resources, additional speedup can be obtained for up to $n$ MPI processes at the cost of some efficiency. As a result of the design of the MPI version the code will not scale beyond $n$ processors. To allow additional speedup, OpenMP directives were inserted to exploit loop level parallelism.

## 4.2 Dual Parallelism Scalability

The addition of threads raises the upper limit on the scalability, while maintaining good efficiency. Table 2 gives the execution times for the same 36-wave runs executed under the same conditions. However, the OpenMP directives were actived and the code was run with 2 threads per MPI process. With the addition of the OpenMP directives, the code continues to scale out to 72 processors, as compared to 36 for the MPI version. This raises the maximum attainable speedup of 24, for the MPI version, to 41.8 with only 2 threads per MPI process. The increased speedup comes with a reasonable drop in efficiency.

Execution Times for the MPI/OpenMP version
on an IBM Power3 SMP

| MPI Processes | Total Processors | Execution Time | Speedup | Efficiency |
|---|---|---|---|---|
| serial | 1 | 13h15m | 1 | 100% |
| 2 | 4 | 4h1m | 3.3 | 82% |
| 3 | 6 | 2h44m | 4.8 | 81% |
| 6 | 12 | 1h25m | 9.4 | 80% |
| 12 | 24 | 45min | 17.7 | 74% |
| 18 | 36 | 32min | 24.8 | 69% |
| 36 | 72 | 19min | 41.8 | 58% |

Table 2: Execution times, speedup, and efficiency for various numbers of MPI processes spawning 2 threads each on the IBM Power3 SMP located at ERDC MSRC.

Figure 1 shows a graphical representation of the speedup of both the MPI and the MPI/OpenMP versions. It is clear that both versions provide nearly linear speedup, with the MPI-only version providing slightly better speedup for small numbers of processors. However, for more than 24 processors, the MPI/OpenMP version produces a better speedup than the MPI-only version. Also, the threaded version extends the nearly linear scalability beyond the

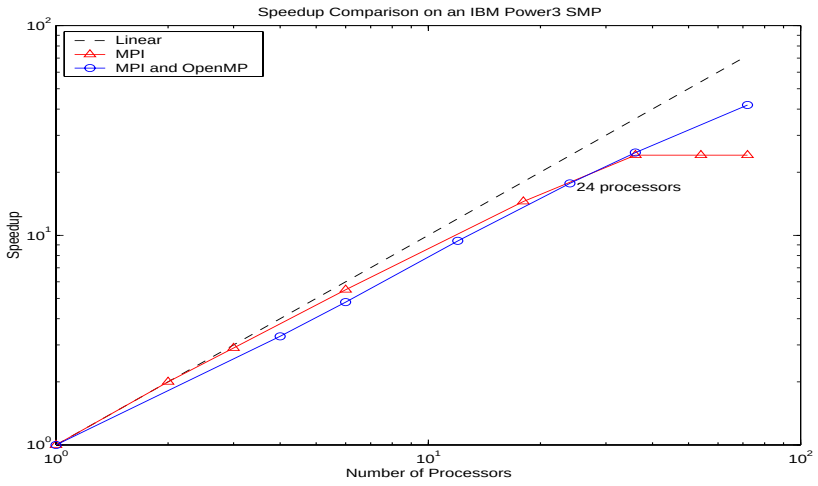


Figure 1: Comparison showing the speedup of the MPI and the MPI/OpenMP version for the 36-wave run test case on the IBM Power3 SMP located at ERDC MSRC. With the MPI version, each processor ran an MPI task. However, with the MPI/OpenMP version only half of the processors ran an MPI task with the others only being utilized for threads.

36-processor point where the MPI version cesses to scale.

As further verification of the parallelization method, each run was repeated on an SGI Origin 2000 located at ERDC MSRC. The results were similar as seen in Table 3. The slower processor speed on the SGI Origin 2000 results in longer execution times and the efficiency is not quite as high as was seen on the IBM Power3 SMP. However, the dual-level program is clearly a viable method of obtaining additional speedup on this machine as well.

Execution Times for the MPI/OpenMP version
on an SGI Origin 2000

| MPI Processes | Total Processors | Execution Time | Speedup | Efficiency |
|---|---|---|---|---|
| serial | 1 | 14hr21min | 1 | 100% |
| 2 | 4 | 4hr33min | 3.2 | 79% |
| 3 | 6 | 3hr10min | 4.5 | 76% |
| 6 | 12 | 1hr38min | 8.8 | 73% |
| 12 | 24 | 51min | 16.9 | 70% |
| 18 | 36 | 35min | 24.6 | 68% |
| 36 | 72 | 22min | 39.1 | 54% |

Table 3: Execution times, speedup and efficiency for various numbers of MPI processes spawning 2 threads each on the SGI Origin 2000 located at ERDC MSRC.

Although only two threads were used per MPI processes in the test case, more threads could have been used to gain additional speedup. In general, the number of threads that can be used is limited by the grid size. In this particular problem, additional threads, up to 7, will continue to yield additional speedup. This was consistent for both machines tested, as was the amount of speedup delivered by threading. This is shown by Table 4.

From the analysis of this test case, dual-level parallelism is shown to be advantageous in two ways. First, threads can provide additional speedup while preserving efficiency. From a comparison of 2 with 1, notice that running this test case with 18 MPI processes spawning 2 threads is more efficient and produces a better speedup than running 36 MPI processes. Thus, the same 36 processors yield better speedup with the MPI/OpenMP parallel code than with the MPI code. Second, when speedup is a high priority, the dual-level parallel code has a much higher maximum speedup. For this test case, the maximum speedup of the MPI version running on the IBM Power3 SMP is 24, but the maximum speedup of the MPI/OpenMP version is 94 when run with 36 MPI processes spawning 7 threads. This run decreased the 13 hours 15 minutes required by the serial run to only 8.5 minutes.

Speedup in Execution Times for Various Numbers of
Threads on an IBM Power3 SMP and an SGI Origin 2000

| Number of Threads | Speedup | |
|---|---|---|
| | IBM Power3 SMP | SGI Origin 2000 |
| 1 | 1.0 | 1.0 |
| 2 | 1.7 | 1.6 |
| 3 | 2.0 | 2.0 |
| 4 | 2.4 | 2.4 |
| 5 | 2.6 | 2.4 |
| 6 | 2.9 | 2.7 |
| 7 | 3.0 | 3.1 |
| 8 | 3.0 | 3.1 |

Table 4: Speedup in execution times for various numbers of threads on the IBM Power3 SMP and SGI Origin 2000 located at ERDC MSRC.

# 5 Conclusion

The MPI version of STWAVE scales out to the number of wave runs being executed. For small numbers of processors, the test case scales linearly, with the time required to process the data being nearly $T_1/p$ where $T_1$ is the time required by the serial program and $p$ is the number of MPI processes used. As the number of processors approaches the number of wave runs, the required time begins to decrease more slowly as the overhead associated with the replicated data becomes a larger factor. For the test case, the maximum speedup of 24 with the MPI version is attained with 36 MPI processes. However, the MPI/OpenMP version running on the same 36 processors with 18 MPI processes spawning 2 threads each exceeds that to yield a speedup of 24.8. In addition, when speedup is a priority, additional speedup can be attained with the MPI/OpenMP version. The amount of speedup that can be gained by threading is dependent upon the grid size of the problem. However, with this test case, the dual-level parallel version yields a maximum speedup of 94 on the IBM Power3 SMP as compared to 24 for the MPI-only version on the same machine.

# References


[1] Foster, Ian. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley, 1995.

[2] Smith, Jane McKee, Donald T. Resio, Alan K. Zundel (1999). "STWAVE: Steady-State Spectral Wave Model, Report 1: User's Man-

ual for STWAVE Version 2.0," Instruction Report CHL-99-1, Engineer Research and Development Center, Vicksburg, MS.